STACK MEMORY PROTECTION

TECHNICAL FIELD

The present invention relates in general to managing stacks within a computer system, in particular managing memory used for program stacks and processor stacks.

BACKGROUND INFORMATION

Computer systems use stacks which may be registers or memory blocks where data is stored and retrieved in a particular fashion (e.g., last in first out or first in first out). The operating system (OS) within the computer system may manage numerous stacks, interrupt stacks, kernel stacks, thread stacks, user stacks, etc. Some stacks in a computer system are fixed in hardware (e.g., fixed register stacks) and others are merely memory allocations (stack memory) wherein a block of memory is defined as a stack whose size and the location may be variable. A fixed register stack is easier to manage since the size is known and simple techniques of adding and subtracting entries from a counter give an indication of the availability of locations within the register stack for storing data. When a stack may be variable in size and location then, it may prove difficult to manage the stack memory along with other necessary memory allocation management without wasting (by blocking access) memory space.

Some computer architectures (e.g., Intel IA64) have begun to use multiple variable stacks per execution context compounding the problem of managing the variable stacks within the other memory management requirements. Within stack memory (memory designated by the operating system only for stack use) there have traditionally been two types of stacks, a "program stack" and a "processor stack" (e.g., an IA64 register stack). The program stack is the "traditional" run-time stack where the program saves/restores hardware register data. The processor stack is a new additional run-time stack required by some computer architectures (e.g., IA64) so the processor can save/restore hardware register data, however these stacks may be transparent to a compiler or a programmer. The operating system (OS) maintains tables of available memory for the processor to use as a processor stack. The main difference between the program stack and the processor stack is that the program stack is used by the compiler for procedural local data and the processor stack is used by the processor for stacked register data (essentially making the registers a cache of recently used, current context registers) and may be transparent to a programmer (e.g., in IA64).

One of the classic issues in managing stack memory is how to protect against a stack over run or under run. Traditionally, protection against an over run or an under run has been solved by allocating "red zones" or protected pages in memory. These protected pages were set up to cause a fault when an over run or under run occurs. The problem (which may be compounded with multiple variable stacks per execution context) in some computer architectures (e.g., IA64) is that pages are

wasted in the requirement to set up the "red zones" or protected pages for every stack in the system. Also of concern is the fact that "red zones" do not always work, for example, a stack pointer may be decremented an amount greater than a "page size" and effectively skip over the protected page or "red zone". Likewise, the future use of multiple variable stacks per execution context (e.g., IA64) may create stacks that are potentially transparent to the compiler or programmer. Of particular concern are operations within these variable program stacks where information is corrupted without a corresponding error condition being signaled so that error correction techniques would enable recovery.

There is therefore a need for a system and method for managing stack memory that prevents stack corruption without there being appropriate recovery in place.

SUMMARY OF THE INVENTION

A new form of memory protection classification is implemented specific for "stack memory". This memory protection classification is implemented in a central processor (CPU) which is supported by a compiler and an operating system. The CPU, in creating the protection classification, generates a page table entry attribute wherein each page of a stack is mapped with the page table entry attribute. Compilers supporting the page table entry attribute would generate specific forms of load/store instructions when saving or restoring to/from program stacks. These new "stack" memory load/store instruction forms would result in the memory references requiring "stack memory protection" to succeed (assures that the stack memory load or store is executable) or a fault is raised. For all stack uses where stacks may be transparent to the compiler or the programmer, the references are generated internally by the CPU such that stack memory attributes are set for the processor stack. The stack memory protection results in both errant normal loads and stores to stack memory and errant stack memory loads and stores to non-stack memory causing faults. Multiple stack memory attributes are generated which differentiate memory into three dynamic regions. Assigning stack memory with a program stack attribute or a processor stack attribute leaves all remaining memory not so designated free for normal loads and stores. Flagging of error conditions if load/stores are attempted into unauthorized regions enables stack memory protection without wasting valuable memory space with variable protected pages where error detection may be unreliable. Protected

pages that have to be speculatively set-up and managed by the operating system are no longer required.

The foregoing has outlined rather broadly the features and technical advantages of the present invention in order that the detailed description of the invention that follows may be better understood. Additional features and advantages of the invention will be described hereinafter which form the subject of the claims of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention, and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

FIG. 1 is a block diagram of CPU hardware circuits used in embodiments of the present invention;

FIG. 2 is a flow diagram of method steps used in embodiments of the present invention;

FIG. 3 is a block diagram of a data processing system which comprises a CPU which has an architecture, compiler and an operating system which may use embodiments of the present invention;

FIG. 4A and FIG. 4B are block diagrams of some prior art program stacks employing protected pages; and

FIG. 5 is a block diagram of memory employing embodiments of the present invention.

DETAILED DESCRIPTION

In the following description, numerous specific details are set forth such as specific word or byte lengths, etc. to provide a thorough understanding of the present invention. However, it will be obvious to those skilled in the art that the present invention may be practiced without such specific details. In other instances, well-known circuits have been shown in block diagram form in order not to obscure the present invention in unnecessary detail. For the most part, details concerning timing considerations and the like may have been omitted in as much as such details are not necessary to obtain a complete understanding of the present invention and are within the skills of persons of ordinary skill in the relevant art.

Refer now to the drawings wherein depicted elements are not necessarily shown to scale and wherein like or similar elements are designated by the same reference numeral through the several views.

FIGS. 4A and 4B illustrate a prior art memory block 401 designated by an operating system (OS) as stack memory for use as program stacks or processor stacks. Memory block 401 is shown with two different allocations made by the OS before executing a program. In FIG. 4A, memory block 401 is allocated as 50% for program stacks 402 and 50% for processor stacks 403. The OS decides before running a program how memory block 401 is to be allocated to program stacks 402 and processor stacks 403. The OS has two ways in which to assign individual stacks in their respective areas. The OS may start by first assigning program stacks 402 to the

higher memory addresses 405 and the processor stacks 403 to the lower addresses 406. As the space in memory block 401 is dynamically assigned, the regions defined by higher addresses 405 and lower addresses 406 "grow" toward each other. To prevent program stacks or processor stacks from over running, a "red zone"or protected page 407 would have to be placed between the program stack 402 area and the processor stack 403 area. In FIG. 4B, the OS has allocated 25% of the memory block 401 to program stacks 402 and 75% to processor stacks 403. In FIG. 4B, the OS assigns the starting addresses so the two stack regions (405 and 406) "grow" apart. In this case two different protected pages 408 and 409 are placed at the memory block bounds (lower and upper). If an access is attempted to a protected page, an error would be indicated. However, there are cases where decrementing a pointer may skip over a protected page (e.g., 407 in FIG. 4A) and an errant write or read may occur in unprotected memory space without a fault indication that a program stack is being written in a processor stack region.

FIG. 1 is a block diagram of load/store hardware 100 within a CPU (see CPU 310 in FIG.3) used in embodiments of the present invention. Instructions 103 are sent to a load/store unit 101. Decoding load/store instructions will generate a load/store operation 104 which determines whether data is to be written into or read from memory 108. Translation hardware 111 looks up a corresponding virtual address (translation 112) in memory page attribute table 105 to determine if a memory stack attribute 109 is present in the memory block (not shown) being accessed. Address bus 106 sends the actual physical addresses to the memory 108 for the accessed

memory block. In embodiments of the present invention, the load/store unit 101 may receive different types of load/store instructions. If the operation is a store to stack memory (within memory 108), then the OS of the CPU 310 generates a stack memory attribute 109 which is associated with the memory block used to store data 102 via data bus 107. In a subsequent stack memory load, the CPU 310 will determine if the memory address requested is associated with the stack memory attribute 109. If the required stack memory attribute 109 is present, then the load is allowed to proceed. If the required stack memory attribute 109 is not present an error is flagged in error trap 110.

FIG. 5 illustrates a block of memory 500 utilizing embodiments of the present invention. AProg 507 illustrates a stack memory attribute associated with program stack 504. Program stack 504 is a block of memory defined with stack memory load and store instructions and a stack memory attribute (AProg 507) identifying the stack memory as a program stack. AProc 508 illustrates a stack memory attribute associated with processor stack 506. Normal memory blocks (e.g., 501, 502, 503 and 505) do not have a stack memory attribute (e.g., AProg 507 or AProc 508) and are only useable for normal loads and stores. Likewise, memory stack blocks 504 and 506 are only useable for stack memory loads and stores. In particular, memory stack block 504 has a program stack attribute (AProg 507) and is useable only for program stacks, and memory stack block 506 has a processor stack attribute (AProc 508) and is useable only for processor stacks.

FIG. 5 illustrates that managing stack memory, using embodiments of the present invention, does not require any "red zones" or protected pages. The OS of the CPU 310 associates a stack memory attribute in a page table (not shown) with each block of memory accessed using a stack memory load/store instruction. The OS no longer has to make an estimate of how it is going to allocate a portion of memory designated as program stack. Rather, stack memory attributes are associated with memory used as stacks (program or processor stacks) on an "as needed" basis thus saving memory space. It should be noted that other stack memory attributes or memory attributes may be defined from managing different types of loads and stores to memory and still be within the scope of the present invention.

FIG. 2 is a flow diagram 200 of method steps used in embodiments of the present invention. In step 201, a load/store instruction is decoded in an execution unit. In step 202, the instruction is tested to determine if the instruction is a normal load/store instruction. If the result in step 202 is YES, then a branch to step 214 is executed to determine if the memory page addressed has a stack memory attribute. If the result of the test in step 214 is YES, then an error is flagged in step 215 and a return to step 201 is executed. If the result of the test in step 214 is NO, then in step 218 a normal load/store is executed and a return to the execution unit is executed in step 216. If the result of the test in step 202 is NO, then a branch to step 203 is executed. In step 203 a test is executed to determine whether the memory stack instruction is a stack memory load. If the result in test 203 is NO, then a test is executed in step 220 to determine if the memory page exists. If the result of the test

in step 220 is NO, then in step 210 the memory page attribute is stored designating the memory block as stack memory. In step 209, the data is stored in the memory block and a return to step 201 is executed in step 217. If the result in step 220 is YES, then a branch is executed to step 211 to determine if the memory page has the required stack memory attribute. If the result of the test in step 211 is YES, then in step 212 the data is stored in the memory block and in step 213 a return is executed to step 201. If the result of the test in step 211 is NO, then an error is flagged in step 219 along with a return to step. If the result of the test in step 203 is YES, then a test is done in step 204 to determine if the memory page being requested has a stack memory attribute. If the result of the test in step 204 is YES, then data is read using stack protocol in step 208. If the result of the test in step 204 is NO, then access to the memory block is prevented in step 205 and a return to step 201 is executed in step 206.

Referring to FIG. 3, an example is shown of a data processing system 300 which may use embodiments of the present invention. The system has a central processing unit (CPU) 310, which is coupled to various other components by system bus 312. Read-Only Memory ("ROM") 316 is coupled to the system bus 312 and includes a basic input/output system ("BIOS") that controls certain basic functions of the data processing system 300. Random Access Memory ("RAM") 314, I/O adapter 318, and communications adapter 334 are also coupled to the system bus 312. I/O adapter 318 may be a small computer system interface ("SCSI") adapter that communicates with a disk storage device 320 or tape storage device 340. A

communications adapter 334 may also interconnect bus 312 with an outside network 341 enabling the data processing system 300 to communicate with other systems. Input/Output devices are also connected to system bus 312 via user interface adapter 322 and display adapter 336. Keyboard 324, trackball 332, mouse 326, and speaker 328 are all interconnected to bus 312 via user interface adapter 322. Display 338 is connected to system bus 312 via display adapter 336. In this manner, a user is capable of inputting to the system through the keyboard 324, trackball 332, or mouse 326, and receiving output from the system via speaker 328 and display 338.

CPU 310 may execute software programs under an operation system using stack memory load/store instructions according to embodiments of the present invention where stack memory attributes are assigned in a page table. These stack memory attributes (e.g., processor or program) allow memory blocks to be designated for use as specific types of stacks on an as needed basis. Using embodiments of the present invention, an operating system running on CPU 310 would no longer have to speculatively allocate memory as stack memory saving memory space and improving stack memory protection.